



20450 Century Boulevard
Germantown, MD 20874
Fax: (301) 515-7954

FFTC Driver

Software Design Specification (SDS)

Revision A

8/25/10

Document License

This work is licensed under the Creative Commons Attribution-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nd/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Contributors to this document

Copyright (C) 2011 Texas Instruments Incorporated - <http://www.ti.com/>

Revision Record	
Document Title: Software Design Specification	
Revision	Description of Change
A	<ol style="list-style-type: none"> 1. Initial Release (Release v 1.0.0.0) 2. Replaced code snippets with Illustrative figures; Added “Porting Guidelines” section; Added “OS Abstraction Layer API” section. (Release v 1.0.0.1) 3. Changed title of the document to “FFTC Driver”; Separated “Flow objects” into “Rx” and “Tx” objects in driver design. Added support for multiple FFTC peripheral instance management. Updated the doc to reflect this. (Release v 1.0.0.2) 4. Documented interrupt and FFTC multi-instance support in the driver. (Release v 1.0.0.3) 5. Added documentation for new features: “Interrupt configuration on per Rx object basis” and “Flow sharing between multiple Rx objects”. (Release v 1.0.0.4/v 1.0.0.5) 6. Updated documentation to reflect on CPPI tag usage by the driver, removed Tx queue threshold configuration. (Release v 1.0.0.6) 7. Added new OSAL cache APIs. Updated documentation to reflect following feature changes: Core level interrupt setup removed; Ability to submit and process application owned descriptors added; Expose complete flow/FDQ and accumulator management to the application. (Release v1.0.0.9) 8. Added a new set of OSAL APIs for locking/unlocking interrupts and for data buffer cache synchronization. Renamed the multicore OSAL APIs. (Release v1.0.0.10)

Note: Be sure the Revision of this document matches the QRSA record Revision letter. The revision letter increments only upon approval via the Quality Record System.

TABLE OF CONTENTS

1	SCOPE.....	1
2	REFERENCES.....	2
3	DEFINITIONS	3
4	OVERVIEW.....	4
5	DESIGN.....	5
5.1	GOALS.....	5
5.2	OVERVIEW.....	7
5.3	CPPI/QM USAGE.....	8
5.4	DRIVER TERMINOLOGY	11
5.5	INTERNAL DATA STRUCTURES.....	14
5.6	EXTERNAL DRIVER APIS	16
5.7	FFTC LLD API USAGE.....	16
5.8	FFTC HIGHER LAYER API USAGE	16
5.8.1	<i>Initialization.....</i>	<i>16</i>
5.8.1.1	System Level Initialization	16
5.8.1.2	Application Level Initialization	17
5.8.2	<i>Transmit Path.....</i>	<i>24</i>
5.8.2.1	Submit using Driver Managed Tx Descriptors	24
5.8.2.2	Submit using Application Managed Tx Descriptors	27
5.8.3	<i>Receive Path</i>	<i>27</i>
5.8.4	<i>De-initialization</i>	<i>33</i>
5.9	FFTC OS ABSTRACTION LAYER APIS.....	33
5.9.1	<i>Memory Management.....</i>	<i>34</i>
5.9.2	<i>Multi-core Synchronization</i>	<i>34</i>
5.9.3	<i>Interrupt Lock</i>	<i>35</i>
5.9.4	<i>Logging.....</i>	<i>36</i>
5.9.5	<i>Process notification.....</i>	<i>37</i>
5.9.6	<i>Cache Synchronization.....</i>	<i>38</i>
5.10	FFTC DEVICE SPECIFIC LAYER APIS	40
6	PORTING GUIDELINES.....	40
6.1	DATA TYPES.....	41
6.2	OS AL APIS	41
6.3	CONFIGURATION OPTIONS	42
7	FUTURE EXTENSIONS	43

1 Scope

This document describes the design and usage of the FFTC driver and its APIs.

2 References

The following references are related to the feature described in this document and shall be consulted as necessary.

No	Referenced Document	Control Number	Description
1	FFTC_Users_Guide.pdf	Version 1.0.5 June, 2010	FFT Co-processor Users Guide
2	CPPI_Users_Guide.pdf	Version 0.5.7 June, 2010	TMS320TCI66xx Multicore Navigator (CPPI)

Table 1. Referenced Materials

3 Definitions

Acronym	Description
API	Application Programming Interface
FFTC	Fast Fourier Transform Co-processor
CPPI	Communications Port Programming Interface
LLD	Low Level Driver
HLD	High Level Driver
QMSS	Queue Manager Sub System
LTE	Long Term Evolution
AIF	Antenna Interface
OSAL	OS Abstraction Layer
Tx	Transmit
Rx	Receive

Table 2. Definitions

4 Overview

The Fast Fourier Transform Co-processor (FFTC) is a hardware co-processor that can be used to offload and accelerate Fast Fourier Transform (FFT) and Inverse FFT (IFFT) computations that are required by various DSP applications. It's been designed to be compatible with various OFDM based wireless standards like WiMax and LTE.

The FFT co-processor hardware consists of the following high level sub-modules:

1. *Configuration Registers:*

Useful in specifying the configuration to be applied for performing the DFT/IDFT computation on data, reading the error and status bits from the FFT computation engine.

2. *FFT Engine:*

Performs the actual DFT / IDFT calculations on the data handed over to it.

3. *CPPI DMA:*

The Input-Output (I/O) interface for the FFTC engine, i.e., handles all the data movement in and out of the FFTC without any DSP intervention.

4. *FFTC Streaming Interface:*

Links the CPPI-DMA and the FFTC engine and manages the handshake and byte ordering issues between them.

5. *FFTC Scheduler:*

The FFTC CPPI-DMA exposes multiple transmit queues to the FFTC engine. i.e., data can be buffered at and sent from any one of these queues to the FFT engine for processing. The FFT scheduler is responsible for prioritizing which of the data packets from these transmit queues needs to be processed next.

The FFT engine is very closely coupled with the CPPI/QM Subsystem and the only way to send FFT requests and receive the results would be by using the CPPI/QM subsystem. This puts the additional burden on an FFTC user, i.e., any application or driver writer that uses FFTC to also understand the working and configuration of CPPI/QMSS apart from understanding the FFTC configuration.

The FFTC driver is an attempt at simplifying the interface to the FFTC user such that only the essential CPPI/QM details are exposed. This document covers the design goals, assumptions made, APIs and data structures exported by the FFTC driver to this extent.

This document assumes that the reader has familiarized him/herself with the FFTC User's Guide, and the CPPI/QM specifications briefly.

5 Design

5.1 Goals

The design goals for the FFTC driver can be briefly summarized as follows:

1. *Provide basic low performance impact APIs:*

Provide APIs at various levels of granularity, i.e., expose a higher layer abstract, task oriented APIs (for example, to configure FFTC queues, submit FFT requests, parse the result etc) and also a lower layer set of LLD APIs that can be used to quickly configure the hardware without much performance impact.

2. *OS independent:*

The driver should be designed to be OS independent to ease porting from one OS to another.

3. *Multi-core aware:*

The FFT co-processor hardware is designed to be accessible across multiple cores in a multi-core chip. Hence, the FFTC driver should be aware of the multi-core challenges and must provide for mechanisms to address them.

Keeping the design goals in mind, the FFTC driver is organized as follows:

1. *Higher Layer APIs:*

The FFTC higher layer exports APIs that can be used to initialize the FFTC engine, configure the FFTC Tx queues, setup CPPI receive flows, to submit FFT requests, and finally get and parse the FFT results from the FFT engine.

These APIs in turn use:

- CPPI and QMSS library APIs to setup the FFTC CPPI DMA
- FFTC Lower layer (LLD) APIs to setup any FFTC registers required and to format the configuration provided by the application to FFT headers that the hardware understands.

- OS Abstraction layer (AL) APIs to keep up with OS independence
- Device (SoC) specific API call-outs to obtain SoC specific configuration such as QM accumulator interrupt channel, queue number to use etc.

2. *OS Abstraction Layer (OS AL):*

The OS AL defines APIs for memory management, multi-core and process synchronization. Separating these functions out of the FFTC driver simplifies porting of the LLD between different OS.

3. *Device Specific Layer:*

The device specific layer implements the APIs required by the driver to retrieve SoC specific configuration such as QM accumulator interrupt channel, queue number to use.

4. *Low Level Driver (LLD) APIs:*

This layer exposes various APIs useful in reading and writing values from/to FFTC memory mapped registers. It also exports various utility APIs that can be used by an application/driver to format user specified configuration data structures to FFT engine readable header formats. The FFTC Lower layer APIs in turn use the FFTC CSL Register Layer (RL) file from the CSL package for FFTC memory mapped register overlay.

The FFTC driver organization and its dependencies are as shown below:

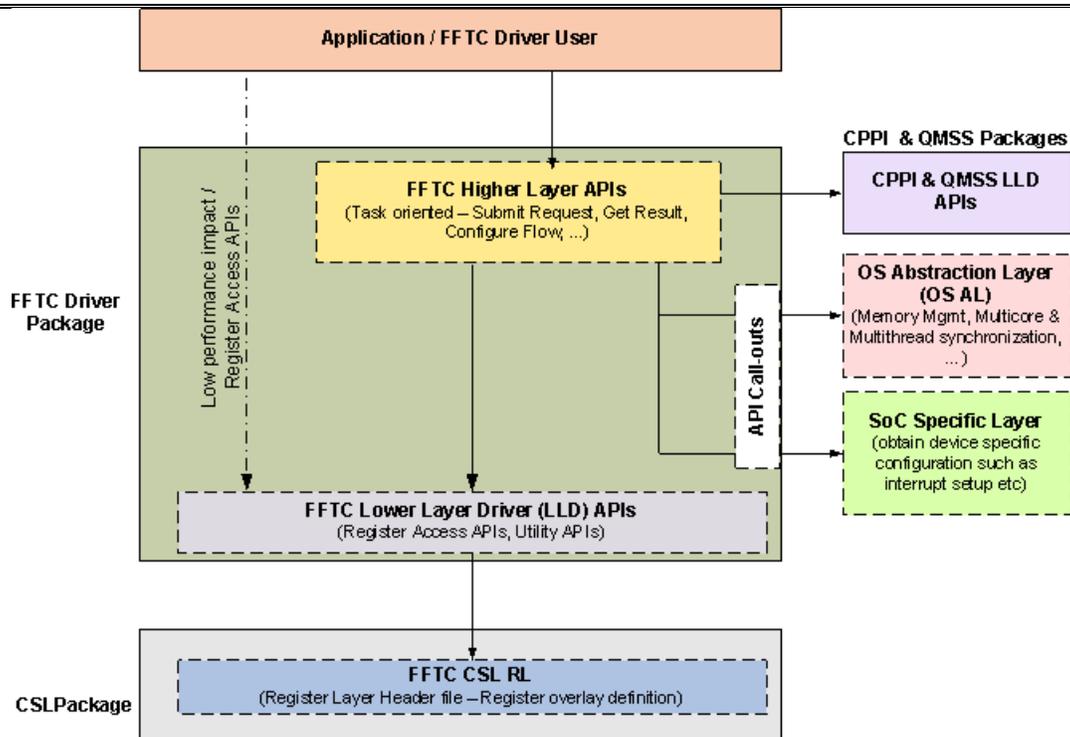


Figure 1. FFTC Driver Organization

5.2 Overview

The FFTC driver's functions can be categorized as follows:

1. *FFTC peripheral management:*

The driver initializes and manages configuration of multiple FFTC peripheral instances on the SoC. It manages each of the FFTC instance's various shared resources such as configuration registers, interrupts etc in a consistent manner to achieve predictable, and reliable output from the hardware every time. It treats each FFTC peripheral instance independent of each other and manages each of them separately.

2. *CPPI resource management:*

The driver manages all CPPI resources if requested such as CPPI channels, flows, buffer descriptors, etc internally, exposing only relevant configuration to application.

3. *QMSS queue management:*

It also sets up and manages if requested all the hardware queues required by FFTC driver users to send and receive data to/from the engine.

Rest of the document covers the internals of the driver design from a single FFTC instance perspective. The same discussion applies to each of the other instances of FFTC since they are all identical and have no interactions with each other.

5.3 CPPI/QM Usage

The CPPI-DMA (briefly written as CPDMA) in the FFT co-processor architecture enables the FFT computational engine to be shared across multiple applications running on various cores in an effortless manner. Towards this end, 4 CPPI/QM queues and 8 CPPI flows have been allocated for each of the FFTC instance's CPDMA in the system.

We define the following the two commonly used CPPI/QM terminologies in the FFTC driver here:

1. *FFTC Transmit queues:*

- a. The FFTC transmit (Tx) queues are Queue Manager Subsystem (QMSS) queues that are capable of buffering FFT request data blocks while the FFTC engine processes them and outputs the results to a CPPI destination queue reserved by the application.
- b. There are four such Tx queues in each of the FFTC instance CPDMA.
- c. Each of these Tx queues can be programmed to have different priority levels, thus helping one application's FFT data to be prioritized over another by the FFT engine for computation.
- d. The FFTC scheduler operates on a per-block basis to check if any data is available on a higher priority queue to service. The scheduler works on a "*starvation prevention round-robin scheme*" basis.
- e. Each of these queues is characterized using the "*FFTC Queue specific configuration*", i.e., each queue can have a separate configuration that specifies to the FFTC engine how the data arriving on it must be processed.
- f. The configuration for each of the FFTC Tx queues can be specified using two possible ways, firstly by directly programming the queue specific registers and secondly by specifying the queue configuration using CPPI packets to the engine.

The FFTC driver presents these 2 different ways of queue configuration using 2 different modes of queues:

i. Dedicated Mode:

- In this mode, the configuration for the Tx queue can be specified during the queue open time and ONLY once.
- At this time, the five queue specific configuration registers are programmed with the configuration provided. This configuration is then used for all the FFT requests for the lifetime of the application.
- The queue is marked as a “*dedicated*” resource, i.e., it is reserved to be used exclusively by the application that opened it first and with the configuration specified at open time.
- This mode works best for applications whose FFT data have a fixed pattern and hence a fixed configuration too that wouldn’t need to be re-programmed on the go using CPPI packets.
- The demerit of using this mode is that the queue configuration cannot be re-programmed unless the queue is closed and opened again with a different configuration. Also, since there are only 4 queues available in the system for FFTC, not all applications can have the luxury of reserving them exclusively for use in this mode.

ii. Shared Mode:

- In this mode the queue configuration can be specified on the go, i.e., a complete queue configuration can be specified individually for every FFT request packet being sent to the FFTC engine.
- This allows for multiple applications with different FFT requirements to “*share*” the queue, now since each packet can be tagged with a unique configuration to use for processing that specific data packet.
- This scheme works best for applications that do not have a fixed FFT data configuration and hence can specify the configuration on the fly on a per packet basis.
- The demerit of using this mode is that the FFT configuration for a queue MUST be specified with every FFT request being sent to the engine using the driver.

2. FFTC CPPI Flows:

- a. An FFTC CPPI flow defines the destination queue where the FFT result needs to be en-queued to once ready for an application. It also defines the Rx Free Descriptor Queues (FDQ) to use to put together the result.
- b. It also defines the properties of the FFT result packet, such as whether any protocol specific pass through info (PS info) (any special info that needs to be communicated to the FFT data receiver) would have to be put on the FFT result by the engine after processing, if so its location in the packet, the type of CPPI descriptor to use for result buffers, their sizes etc.
- c. There are eight such flows in the FFTC CPDMA.
- d. Each flow is assumed to have a distinct destination queue number.
- e. Unlike FFTC Tx queues whose configuration can be changed at run time using CPPI, the flows cannot be reprogrammed on the fly without affecting data using the flow. All data on the flow would have to be disabled before changing the flow configuration. Hence, a flow cannot be as such re-configured using the FFTC driver, instead it would have to be closed and re-opened with a new configuration if desired.

The following figure illustrates the usage of different modes of CPPI Tx queues and flows in FFT processing.

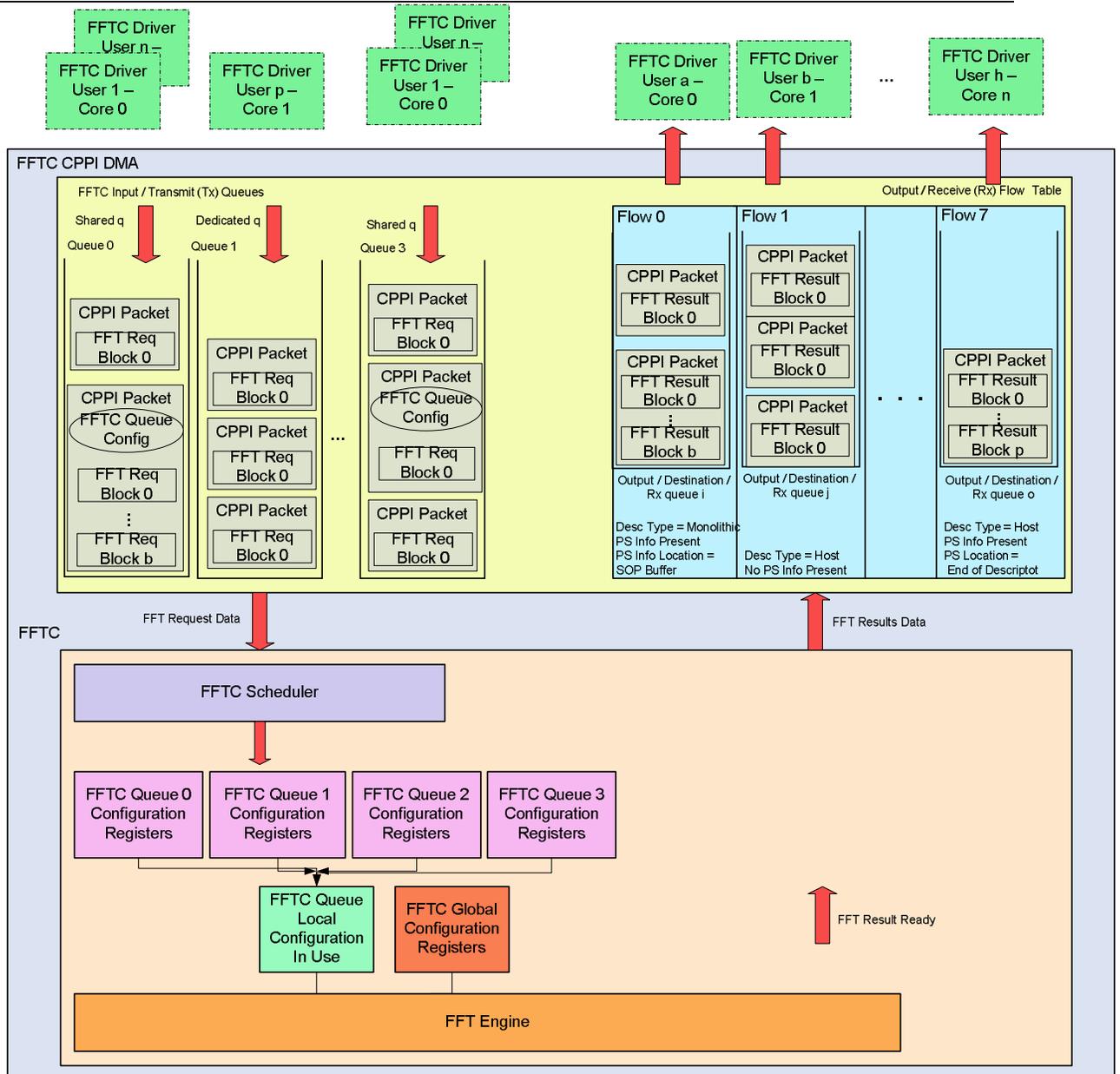


Figure 2: FFT Processing using CPPI Tx Queues and Flows

5.4 Driver Terminology

This section defines the commonly used terms through out the driver and this document.

1. FFTC Transmit Object:

- a. An FFTC transmit object captures all the transmit (Tx) properties of an application that wishes to use the driver to send or submit request data to the FFT engine.

- b. Some properties that characterize a Tx object are:
 - i. FFTC Tx queue number to use for the requests submitted using this object.
 - ii. FFTC Tx queue open mode i.e. if the Tx queue must be used in shared or dedicated mode.
 - iii. If the Tx descriptors are to be allocated and managed by the driver and if so, the descriptor type to use for the requests; Number of such descriptors, maximum size of request data that will be submitted using this object.
 - iv. If the requests submitted using this object would have mixed size DFT blocks or would they all be of the same size?
- c. If the driver was requested to manage the Tx descriptors, it creates a Tx Free descriptor queue (FDQ) for the Tx object and allocates buffers using the OS-AL API call outs and pre-populates the Tx Free descriptors on Tx FDQ with these buffers.
- d. Since the buffers allocated on the Tx object's Tx FDQ can be local to the core's memory, in a multi-core setup its recommended for the application to create at least one Tx object on each of the cores from which it wishes to submit FFT requests using the driver.
- e. One or more Tx objects can be created on any of the cores in the system.

2. FFTC Receive Object:

- a. An FFTC receive object captures all the receive (Rx) properties of an application that wishes to use the driver to receive results from the FFT engine.
- b. Some properties that characterize a Rx object are:
 - i. Rx queue number that the application wishes to retrieve the results from.
 - ii. If the application wishes to use interrupts to receive the result or if it'd like to poll instead.
 - iii. Receive mode, i.e., if using interrupts the application could choose "blocking" or "non-blocking" mode of receive. In "blocking" mode the driver enables the application to block waiting on a result and the driver in turn uses a software semaphore created for the Rx object to notify the application of a result as soon as it arrives.

- iv. Interrupt configuration to use for the Rx object. The driver allows high priority accumulation interrupts to be setup in 2 ways:
 - Driver Managed Accumulator configuration: In this mode, the driver exposes only essential accumulator configuration to the application and driver creates the accumulator list and does the ISR handling for the application if registered.
 - Application Managed Accumulator configuration: In this mode, the driver exposes the complete accumulator configuration to the application. The application is responsible for setting up the accumulator list and its configuration. The driver programs the accumulator with the configuration provided as is. Also the application is responsible for doing the ISR handling for the accumulator its sets up in this mode.
- v. Flow configuration for the Rx object. The driver provides for the Rx flow setup using 2 ways:
 - Driver Managed Flow configuration: In this mode, the driver exposes only a few essential parameters required for Rx flow and Rx FDQ setup. The driver opens an Rx FDQ and allocates buffers using the OS-AL API call outs, pre-populates the descriptors on Rx FDQ with these buffers. It also sets up the Rx flow with some driver assumed defaults and the PS info and location parameters passed by the application. In short, the driver owns the Rx FDQs and flow setup and management in this mode.
 - Application Managed Flow Configuration: In this mode, the complete flow configuration is exposed to the application. The application is expected to setup all required Rx FDQs and setup a valid configuration for the Rx flow. The driver opens the Rx flow with the configuration provided as is.
- c. Since the buffers allocated on the Rx object's Rx FDQ can be local to the core's memory, in a multi-core setup its recommended for the application to create at-least one Rx object on each of the cores from which it wishes to receive FFT results using the driver.
- d. The driver also opens and maintains an Rx queue corresponding to the Rx queue number specified.
- e. The driver expects that there be only one application in the system retrieving results from any given Rx queue.

- f. One or more Rx objects can be created on any of the cores in the system.
- g. The driver allows multiple applications on any given core and FFTC instance to share the same flow and Rx FDQ.

5.5 Internal Data Structures

Internally the FFTC driver maintains a global data base of the following data structures:

1. *FFTC Instance Object:*

- a. This structure is maintained on a per FFTC peripheral instance basis.
- b. Holds the FFTC peripheral instance number.
- c. Reference count to track the number of applications using this instance. Used to ensure that the instance is not de-initialized while in use by some other application.
- d. Maintains the FFTC CPDMA's CPPI Object handle for the corresponding instance number.
- e. Maintains the FFTC LLD handle for the corresponding FFTC instance.
- f. Tracks the usage of DFT size list (used to configure mixed size DFT list in FFTC engine hardware) for this FFTC instance in the driver. This is done so as to ensure that no two applications/Rx objects are using the DFT size list register group at the same time in the system, since this is a shared resource among all the queues in the FFTC hardware.
- g. Tracks relevant state information for all the Tx queues and Rx flows corresponding to this instance.

The following data structures are maintained on a per FFTC peripheral instance basis:

1. *FFTC Tx Queue State Objects:*

- a. There are 4 such object instances maintained in an array referenced by the Tx Queue Number in O (1) time for a given FFTC peripheral instance.
- b. Holds the transmit queue number (can range between 0 – 3 both inclusive) itself.
- c. Hold the Reference counter for this queue object. The reference counter tracks the number of applications that are using the FFTC Tx Queue object. This counter is

incremented every time *Fftc_txOpen ()* API is called and returns successfully and is decremented whenever *Fftc_txClose ()* API is called. When this reference count reaches zero, the transmit queue configuration is cleaned up and this object is considered invalid and cannot be used for submitting FFT requests by an application unless initialized again using *Fftc_txOpen ()* API.

- d. Maintains the CPPI Tx channel and transmit queue handles for this Tx queue object.
- e. Tracks whether this Tx queue was opened in “*shared*” or “*dedicated*” mode.
- f. Stores useful queue configuration such as suppress side info, cyclic prefix addition value etc for “*dedicated*” mode queues. This information is used later to parse the result obtained from the FFT engine.

2. FFTC Flow State Objects:

- a. There are 8 such object instances maintained in an array referenced by the flow Id number in O (1) time.
- b. Holds the CPPI Flow Id itself (can range between 0 – 7 both inclusive).
- c. Reference count on the flow. Reference count for a flow is incremented whenever a Rx object is opened that uses it and is decremented when a Rx object that uses it is closed.
- d. Rx FDQ that the flow is configured to use and the Rx Free descriptor configuration.

3. FFTC Rx Objects State Info:

- a. Tracks all the Rx objects opened in the system.
- b. Rx object Id to identify each of the Rx objects uniquely in the system.
- c. Boolean flag to track whether the Rx object is valid or not. A Rx object is marked valid in the Rx object database when a Rx object is successfully opened and its info is stored in the database. This flag is cleared when the Rx object at the corresponding entry is closed.
- d. Receive queue number and flow Id the Rx object was configured to use.

5.6 External Driver APIs

Please refer to the Doxygen output from the FFTC driver code deliverable for the data structure definitions, API prototypes and their details for both FFTC LLD APIs and FFTC Higher Layer APIs.

5.7 FFTC LLD API Usage

The FFTC low level driver APIs, i.e., FFTC register access and utility APIs are straightforward and hence are not elaborated here. Please refer to the Doxygen comments and example for usage details.

5.8 FFTC Higher Layer API Usage

This section discusses the FFTC Higher layer APIs in detail. Please refer to the Doxygen output and the example code packaged in “*example*” directory for usage details.

5.8.1 Initialization

Following are the steps to be followed in setting up the FFTC driver for FFT processing:

5.8.1.1 System Level Initialization

1. The first step to using the FFTC driver would be to initialize it. This MUST be done only once during the system startup using the API ‘*Fftc_init ()*’.

The inputs to this function are:

- a. Instance Number: The FFTC peripheral number to initialize.
 - b. FFTC Global Configuration: Contains configuration relevant to setup the “*FFTC Configuration Register*”.
 - c. FFTC Device Configuration: Contains device specific configuration for the peripheral instance.
2. Validates input and resets the driver’s global data structures and state info for the instance number specified.
 3. It opens the FFTC CPDMA corresponding to the instance; sets up the global configuration register for the engine and initializes all instance related internal data structures.

5.8.1.2 Application Level Initialization

1. Each application that would like to use the FFTC driver would need to then setup the driver and descriptor configuration for its use by calling '*Fftc_open ()*' API.

The inputs to this function are:

- a. FFTC Instance Number: FFTC peripheral instance number to use for this application.
- b. Descriptor Configuration: Captured in '*Fftc_DrvCfg*' data structure, contains the Tx/Rx buffer descriptors (Host and Monolithic) configuration for the application, i.e., Number of such descriptors, memory region, descriptor type (Host / Monolithic) as specified in the CPPI initial system configuration for FFTC (memory regions must correspond to what was specified during '*Cppi_init ()*', '*Qmss_insertMemoryRegion ()*' and '*Qmss_init ()*' APIs and the number of descriptors must be a subset of what was allocated at '*Cppi_init ()*', '*Qmss_insertMemoryRegion ()*' and '*Qmss_init ()*').

The '*Fftc_open ()*' API does the following:

- i. Allocates and sets up a pool of free descriptor queues (FDQ) for the application according to the configuration specified and initializes them. These free descriptors are later used by the driver in setting up FDQs for various Tx, Rx objects of the application.
 - ii. On success, the API returns a driver handle (to identify the application and the FFTC instance number that it uses) that must be provided in all future calls to the driver APIs.
2. **Tx Initialization:**

This step applies only if the application would like to act as a sender, i.e., be able to submit FFT requests to the FFTC engine.

The application can submit FFT requests to the engine by opening a "*Tx Object*". The Tx object captures all the FFT Tx parameters of the application such as the FFTC Tx queue to use for processing, the type, and number of descriptors, DFT size list configuration etc.

A Tx object can be created by a sending application using "*Fftc_txOpen ()*" API by specifying the driver handle obtained from "*Fftc_open ()*" API and the transmit configuration captured in "*Fftc_TxCfg*" data structure. The configuration details follow:

- a. Choose the transmit queue number (0-3) using which the FFT requests would be submitted.
- b. The transmit queue can be either configured to be in "*shared*" or "*dedicated*" modes. If all the application's FFT requests have a fixed FFT configuration, then the

application can setup the queue's configuration using '*Fftc_txOpen ()*' API at init time and use the queue for submitting FFT requests as data arrives without having to re-program it during the data path. This is called "*dedicated*" mode. On the other hand, if the application desires to provide the FFT configuration for the queue along with the data being submitted on data path, the application can use the queue in "*shared*" mode.

The application can configure the tx queue in "*dedicated*" mode by setting the boolean input parameter '*bSharedMode*' to 0 and in "*shared*" mode by configuring the parameter to be 1.

- c. Specify the queue specific FFT configuration using the data structure '*Fftc_QLocalCfg*'. The queue specific configuration dictates how the FFT requests using the given transmit queue would be processed by the FFT engine.
- d. The application can choose whether it would manage the Tx free descriptors and buffers required to submit FFT requests or if it would like the FFTC driver to do the management for it. This can be indicated using '*bManageReqBuffers*'.
 - If the application chooses to manage the free descriptors and buffers then, it can indicate the same to the driver using '*bManageReqBuffers*' parameter, by setting it to 0. This mode is most suitable for cases where the FFTC engine is receiving data directly from another IP block (e.g. from AIF) where in the Tx descriptors are used from the transmitting IP block's Rx queue rather than having to allocate separately.
 - When this parameter is set to 1, the FFTC driver takes over the Tx request buffer and descriptor management for the application. This mode is most suited when a DSP application wants to submit requests to the engine using driver APIs.
- e. If the application has requested the driver to manage its request buffers and descriptors (by setting '*bManageReqBuffers*' to 1), then it would have to specify the following additional parameters:
 - i. Choose a descriptor type for the FFT requests in parameter '*descType*'. The application can specify '*Cppi_DescType_HOST*' or '*Cppi_DescType_MONOLITHIC*'.
 - ii. *cppiNumDesc* – Number of pre-allocated descriptors with buffers to maintain for the application. This determines the maximum number of outstanding FFT requests that the application can have using this Tx object.
 - iii. *bufferSize* – The maximum FFT data size (in bytes) that will be submitted using this Tx object.

The descriptor configuration specified here must be a subset of the descriptor allocation done at application init time using *Fftc_open ()* API.

- iv. Enable the *bEnableDftSizeListCfg* flag to 1 and setup the maximum number of DFT sizes that will be specified using this Tx object in *maxDftSizeListLen*, if the application chooses to use a mixed size DFT configuration for its data. The *maxDftSizeListLen* is used then to calculate the additional space to allocate for holding DFT size list configuration in request buffer.
- v. Specify if any Protocol specific pass through (PS) information would have to be passed to the receiver.

bPSInfoPresent – Set to 1 to indicate that the FFT request would contain a PS info that needs to be passed through to the FFT result receiver on output from the FFT engine. When this flag is set to 1, the driver ensures that additional space to hold the PS info is allocated.

- iv. Finally, call *Fftc_txOpen ()* API with the above input parameters.

This API sets up the FFTC Tx queue configuration for the corresponding queue if not already open; increments its reference count on success. However, if the queue is already open and was opened in “*dedicated*” mode, this API returns error as the queue cannot be shared between applications. Otherwise, if the queue is already open and in “*shared*” mode then the API just increments the reference counter. The API also opens a Tx free descriptor queue (FDQ) and allocates Tx free descriptors and fills them with buffers if requested (if *bManageReqBuffers* to 1). On success, this API returns a Tx handle that the application must use to submit requests using the driver APIs.

3. **Rx Initialization:**

This step applies only if the application would like to act as a receiver, i.e., be able to receive results from the FFTC engine using the driver APIs

Setup the “*Rx object*” configuration to indicate where the application wants its results to be buffered and the result configuration itself using the data structure *Fftc_RxCfg*.

- a. Fill in the Rx / Destination Queue number where the FFT result should be placed for the application’s FFT data flow in *cppiRxQNum* parameter. If the application has no preference for a specific Rx queue number, it can specify “-1” in which case the driver would pick the next available queue for the destination queue to buffer FFT results.
- b. Specify if a new flow/Rx FDQ must be created or if the application wants to re-use existing flow properties in *useFlowId* parameter.

- i. Set to -1 to indicate that a new flow/Rx FDQ must be created.
 - ii. If the application would like to share existing flow/Rx FDQ properties, then fill in the flow Id to use for this Rx object in *'useFlowId'* parameter. Flow Id for an existing Rx object can be retrieved using *'Fftc_rxGetFlowId ()'* API.
- c. If *'useFlowId'* was set to -1, i.e., if a new flow must be created, the application must then specify if it'd like the driver to manage the Rx FDQs and flow configuration (driver managed mode) or if it would do so (application managed). It can specify this by setting *'bManageRxFlowCfg'* to 1 for driver managed configuration and 0 otherwise.
- d. If *'bManageRxFlowCfg'* was set to 1, then the application must specify the following parameters required for flow setup in *'rxFlowCfg.drvCfg'* (driver managed configuration structure):
- i. Choose a descriptor type for the FFT results in parameter *'descType'*. The application can specify *'Cppi_DescType_HOST'* or *'Cppi_DescType_MONOLITHIC'*.
 - ii. Specify the FFT result buffering parameters:
 - *cppiNumDesc* – The number of Rx free descriptors with buffers to maintain for buffering FFT results for the application. This determines the maximum number of results that will be buffered for the application at any instance of time.
 - *bufferSize* – Number of bytes to allocate for the result buffer. This should have enough room for any Protocol specific words that the application expects to receive.

The descriptor configuration specified here must be a subset of the descriptor allocation done at application init time using *'Fftc_open ()'* API.

- iii. Specify if the application wants to receive and process any Protocol specific pass through (PS) information from the FFT result on output from the FFT engine. If so, the location where to put the PS info in the result packet.
 - *bPSInfoPresent* – Set to 1 to indicate that the application would like to receive PS info in FFT result from the FFT engine.
 - *psLocation* – Indicate where PS info should be placed in the FFT result packet. Set to '0' to indicate that the PS info must be added

to the “Protocol Specific Words” field of the CPPI descriptor and set to ‘1’ to indicate that the PS info should be at start of the result data buffer itself.

However, if the application chooses to manage the Rx FDQs and flow configuration on its own, it can indicate the same to the driver by setting ‘*bManageRxFlowCfg*’ to 0 and specifying the complete flow configuration in ‘*rxFlowCfg.fullCfg*’.

- e. Pick a mode to retrieve results using this Rx object. Possible modes are:
- i. Use Interrupts – The application can choose interrupts for the Rx object by setting ‘*bUseInterrupts*’ to 1.
 - The application must then specify if it would like the driver to manage the accumulator list and ISR or if it would handle it on its own. It can do so by setting ‘*bManageAccumList*’ to 1 if accumulator list and ISR is to be managed by driver and to 0 for application owned ISR handling.
 - If driver managed interrupt configuration was chosen, i.e., ‘*bManageAccumList*’ was set to 1, the application must specify the interrupt configuration in ‘*accumCfg.drvCfg*’.
 - If however the application chooses to manage the accumulator list and interrupts (ISR) by itself, then it can specify the complete configuration in ‘*accumCfg.fullCfg*’ and the driver will program the accumulator with configuration provided as is.
 - Please note that there might be some restrictions on the use of accumulator channels on various cores for each SoC. Please consult the Accumulator firmware specification and Device level specification before picking a channel number to use here.
 - ii. Use Polling – The application can choose to poll on a Rx object for results. It can configure the Rx object for polling by configuring ‘*bUseInterrupts*’ flag to 0. In this case, any Rx queue can be specified for this Rx object.
- f. If the Rx object is setup to use interrupts (*bUseInterrupts* = 1), then the Rx object can be configured to be either in “blocking or “non-blocking” Rx modes. In blocking mode, the driver creates a software semaphore for this Rx object and uses it to notify of a result to an application waiting on it using ‘*Fftc_rxGetResult ()*’ API. Blocking mode is applicable only if ‘*bUseInterrupts*’ was set to 1 and the driver is managing the accumulator. Blocking mode can be enabled on an Rx object by configuring ‘*bBlockOnResult*’ to 1.

- g. Finally call '*Fftc_rxOpen ()*' API with the FFTC Rx object configuration specified above and the driver handle obtained earlier in Step 1 of the initialization sequence.

The '*Fftc_rxOpen ()*' API based on the Rx configuration specified, opens a Rx Free descriptor queue for this object if requested, allocates result descriptors and buffers big enough to hold them and en-queues them to the Rx FDQ. It also sets up the CPPI Rx flow if requested, corresponding destination queue to buffer the FFT results and stores the handles required. If interrupts were requested, this API sets up the interrupts for the Rx object and creates a semaphore if blocking mode was requested.

On success, this API returns an Rx object handle that the application must use for receiving results from the engine using the driver APIs.

Note:

- Rx / Tx objects CANNOT be shared between applications; every application that would like to send or receive data using FFT driver would have to setup its own object. The driver ensures no synchronization on data path when these handles are shared between applications and if proper care is not taken, could lead to misleading results for the application.
- Also the driver does no validation if the same destination (Rx) queue is used by more than one Rx objects. If two or more Rx objects are using the same Rx queue, then it's application's responsibility to ensure that results are retrieved using the correct Rx object for proper operation.

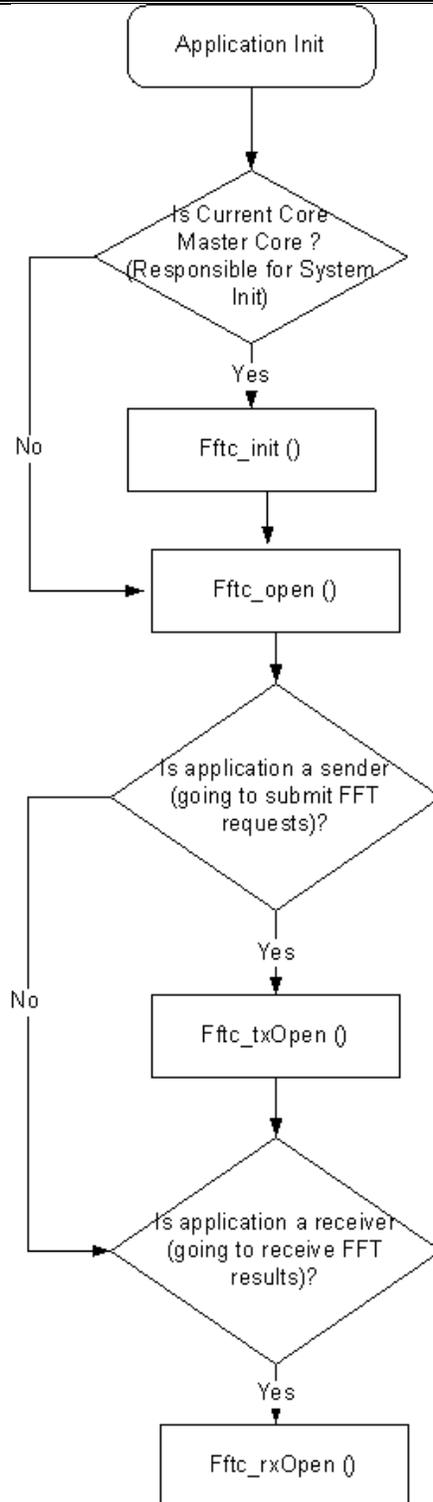


Figure 3: FFTC Initialization Steps

5.8.2 Transmit Path

5.8.2.1 Submit using Driver Managed Tx Descriptors

Steps to follow to prepare and submit an FFT request using the driver allocated Tx descriptors (Tx object must have been opened with *'bManageReqBuffers'=1*) are as follows:

1. Get the request buffer to fill in FFT request data.

If the Tx object was configured such that FFTC driver is to manage the request buffers, then the request buffers have been already allocated for the application by the driver. The application can get this pre-allocated buffer and request handle using *'Fftc_txGetRequestBuffer()'* API.

The inputs to this API are:

- a. Tx object handle
- b. DFT block size information using *'Fftc_BlockInfo'* data structure (specifies whether all DFT blocks of this request are of same size in *'bIsEqualSize'*, list of block sizes if using mixed DFT block sizes in *'blockSizes'* and the number of block sizes specified in *'numBlocks'*).
- c. FFT configuration to use for this request specified using the data structure *'Fftc_QLocalCfg'*
- d. PS info length.
- e. Flow Id that specifies the destination queue where the FFT/IFFT results for the request must be sent to and the Rx FDQ that the FFTC CPDMA must use to put together the result packets. If the application wishes to send the results to an Rx queue other than the one configured on the flow Id specified here, it can do so by specifying the Rx queue number in the FFT configuration *'Fftc_QLocalCfg'* explicitly. If the results are to be sent to the Rx queue configured on the flow, the destination queue number field of the *'Fftc_QLocalCfg'* structure MUST be set to *'0x3FFF'* to indicate the same.
- f. 16 bit Destination Tag Information to put in the descriptor. The application can specify any information that it'd like to use to track the packet or relay from the Tx to Rx end. For example, it can be used to specify a unique Id for the packet that it can associate with the result when received.

This API based on the Tx object configuration, and various configuration parameters

information passed does the following:

- a. Validates the input passed
- b. Gets a Tx free descriptor from the Tx object handle, based on the inputs passed adds a FFTC control header containing the DFT block size and FFT queue configuration and configures the descriptor as much as possible.
- c. On success, this API returns the request handle (CPPI descriptor handle) containing the FFT request settings, the request data buffer in which the application can fill in the PS info/request data and the data buffer length available to the application.

Note:

- The driver expects that the data buffer is filled in the following order ONLY:- PS info if any then followed by the FFT request data itself. PS info should be present before the FFT request data and be filled in the data buffer itself.
- The driver populates the “Source Tag” and “Destination Tag” fields of the descriptor as follows:
 - *Source Tag – High*: This field is not used/set by the driver on Tx.
 - *Source Tag – Low*: This byte field is used by the driver to convey the flow Id to use to process the result for this request packet.
 - *Destination Tag – High*: Set to the higher order 8 bits of the destination tag information passed to the ‘*Fftc_txGetRequestBuffer ()*’ API.
 - *Destination Tag – Low*: Set to the higher order 8 bits of the destination tag information passed to the ‘*Fftc_txGetRequestBuffer ()*’ API.

2. Follow the below mentioned steps to fill in FFT request data:

- a. Get the request data buffer, length and request handle returned by the ‘*Fftc_txGetRequestBuffer()*’ API.
- b. Fill in the PS Info if any to be conveyed to the FFT receiver. Next, fill in all the FFT request blocks ensuring that the PS info length and the total FFT request data length doesn’t exceed the buffer length returned by ‘*Fftc_txGetRequestBuffer()*’ API.
- c. Finally, call the ‘*Fftc_txSubmitRequest()*’ API using the following parameters:

- i. Tx object handle
 - ii. Request handle obtained from '*Fftc_getRequestBuffer ()*' API.
 - iii. FFT request data length in bytes. Must not include the PS info length.
- d. The API returns the appropriate return value to indicate the error status and on success, returns 0.
- e. If the API '*Fftc_txSubmitRequest ()*' returns an error, the application must call '*Fftc_txFreeRequestBuffer ()*' API to free the request handle and descriptor. On success, the request is processed by engine and recycled by hardware automatically and hence the software doesn't need to do anything. Hence, this step is required only if the '*Fftc_txSubmitRequest ()*' returns an error.

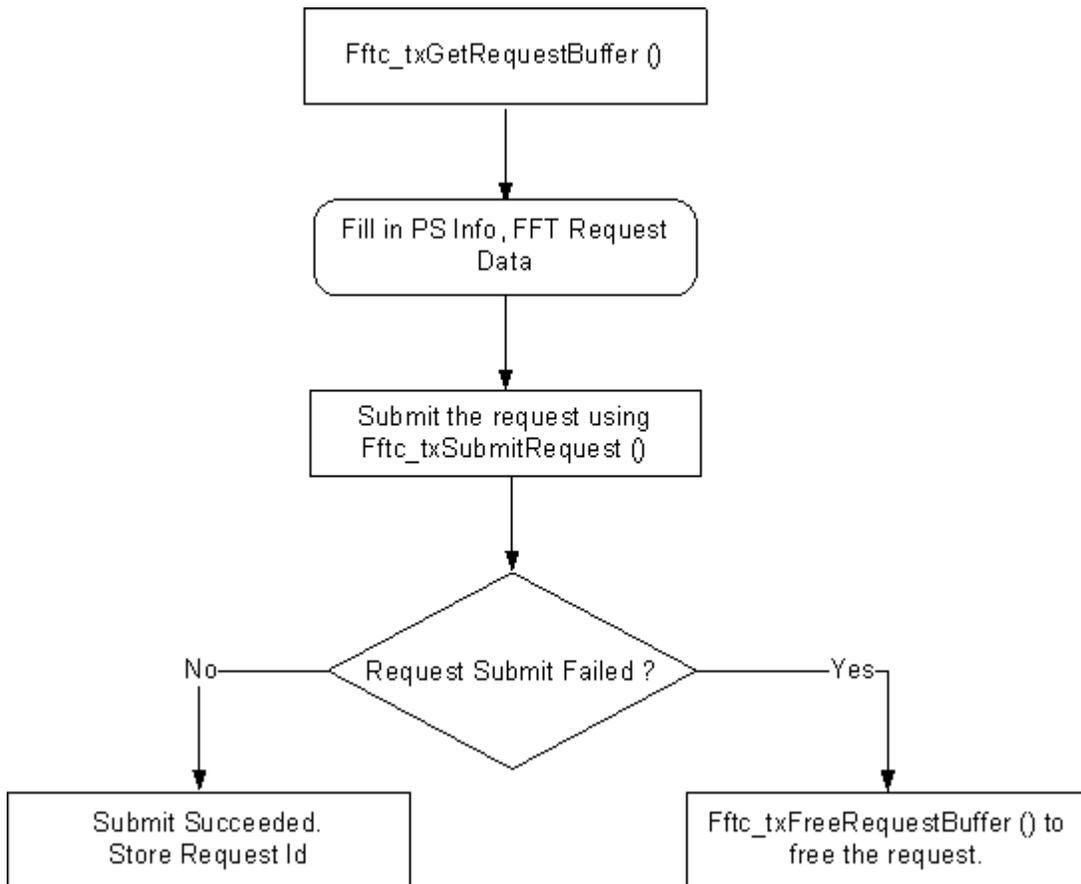


Figure 4: FFTC Request Submission Steps

5.8.2.2 Submit using Application Managed Tx Descriptors

1. If the application is managing its own Tx descriptors, or would like to submit a descriptor received from another module in data path for FFT processing, it can use the API '*Fftc_txSubmitRequest ()*' to submit the descriptor containing the FFT request data.
2. Specify the following to '*Fftc_txSubmitRequest ()*' API:
 - a. Tx object handle. Indicates which FFTC Tx queue to use to submit the request.
 - b. CPPI descriptor handle containing the FFT/IFFT request. This can be either a monolithic/host descriptor.
 - c. FFT request data length in bytes. Must not include the PS info length.
3. The API returns the appropriate return value to indicate the error status and on success, returns 0.
4. If the API '*Fftc_txSubmitRequest ()*' returns an error, the application must free the descriptor. On success, the request is processed by engine and recycled by hardware automatically and hence the software doesn't need to do anything.

5.8.3 Receive Path

The driver supports both High priority accumulation mode interrupts and polling mode.

1. To be enable interrupt processing by the FFTC driver, the application would have to do the following:
 - a. The Rx object on which interrupts are required must have been configured with '*bUseInterrupts*' set to 1.
 - b. Ensure the proper accumulator configuration has been setup either by the driver or by the application using '*Fftc_rxOpen ()*' API.
 - c. If the Rx object was setup such that the driver would manage the interrupts for it, then register '*Fftc_rxHiPriorityRxISR ()*' API exposed by the driver as the FFTC ISR handler with the OS/interrupt management library using the FFTC event id corresponding to the core. The FFTC system event id map can be obtained from the device spec for your EVM. The Rx object handle obtained from '*Fftc_rxOpen ()*' API must be passed as an argument to this ISR handler API.

- d. Instead if the Rx object was setup to use application managed accumulator list and interrupt configuration, then the application should register its own ISR for handling the interrupts and can use the driver API '*Fftc_rxProcessDesc ()*' to process the descriptor received in its ISR. Details of this API can be found this document.
2. If the application wishes to poll on result, then can call the '*Fftc_rxGetNumPendingResults ()*' API from a process context to check on the destination queue to see if a result is available. This API returns the number of results available for the Rx object to process.
3. If the Rx object was setup using driver managed accumulator configuration, the application can retrieve results from the engine using driver APIs as follows:
 - a. The application could retrieve the raw unformatted FFT result from the destination queue using the '*Fftc_rxGetResult()*' API.
 - i. The API returns the pointer to result data buffer pointer, its length, the pointer to PS info pointer, PS info length from the result, a result handle that holds the CPPI descriptor, and the source and destination tag information read from the descriptor.
 - ii. This API returns an appropriate error if any found during its basic error checking.
 - iii. This API is useful if the application wants to quickly get the FFT result without any formatting.
 - iv. This API blocks on the result if the Rx object was configured to use interrupts and to use blocking mode ('*bBlockOnResult*' = 1 and '*bUseInterrupts*' = 1 of Rx object). When the result finally arrives from the ISR this application's Rx object semaphore is posted and the API returns with the result.
 - b. The application could get the FFT result formatted to extract information such as FFT result on a per block basis, any error detected on that block by the FFT engine etc using the API '*Fftc_rxParseResult ()*'.
4. If the Rx object was setup to use application managed accumulator configuration, then the application is responsible for ISR handling. The driver however provides a utility API '*Fftc_rxProcessDesc ()*' that the application can use with the descriptor it received from the engine to retrieve the result buffer, any PS info and source and destination tag information from it.
5. Given a Rx object handle, the CPPI descriptor handle containing the result (or the handle obtained using '*Fftc_rxGetResult()*' API) and request settings such as Sideband

information enable (*'bSuppressSideInfo'*) and Cyclic prefix addition length if cyclic prefix addition was requested, the *'Fftc_rxParseResult ()'* API parses the raw result and breaks it down to an easily understandable result format using the data structure *'Fftc_Result'*.

The formatted result contains:

- a. Flow Id. This is read from the lower order 8 bits of the source tag of descriptor if Side band information is suppressed, otherwise read from the side band information on the packet buffer.
 - b. Source Id. This is read from the higher order 8 bits of the source tag of descriptor if Side band information is suppressed, otherwise read from the side band information on the packet buffer.
 - c. 16 bit Destination tag information. Read from the destination tag of the descriptor if side band information is suppressed, otherwise read from the side band information in the buffer.
 - d. Indicates whether any error was detected by FFT engine processing the result.
 - e. Number of FFT blocks results.
 - f. The FFT result on a per block basis that contains:
 - i. The FFT result buffer itself for the corresponding request block.
 - ii. FFT result buffer length for that block.
 - iii. Block exponent value detected for this block.
 - iv. Indicates whether a clipping error was detected processing this FFT request block.
6. An application can retrieve the number of pending FFT results still to be serviced by the application for any given Rx object using polling or using interrupts managed by driver using the API *'Fftc_rxGetNumPendingResults ()'*.
7. Finally once done processing the FFT result, the application could restore the descriptor using the *'Fftc_rxFreeResult ()'* API.

The application would have to provide the Rx object handle, CPPI descriptor or result handle obtained from *'Fftc_rxGetResult()'* API.

The *'Fftc_rxFreeResult ()'* API returns the PS info buffer and result data buffer to the free result descriptor queue of the Rx object. This API must be called by the application once

it's done processing the result; otherwise the application could run out of free buffers and descriptors for the FFT engine to fill in the results eventually.

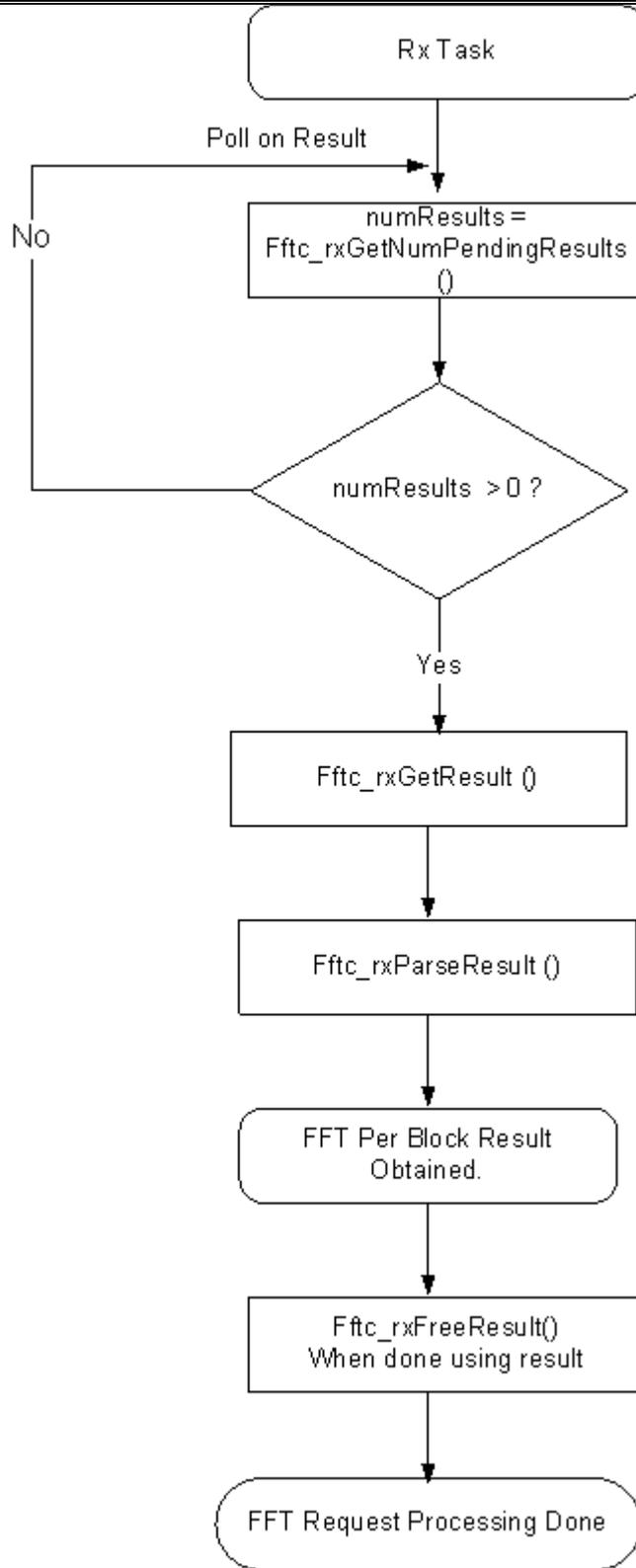


Figure 5: FFTC Receive Path Processing (Polling Mode)

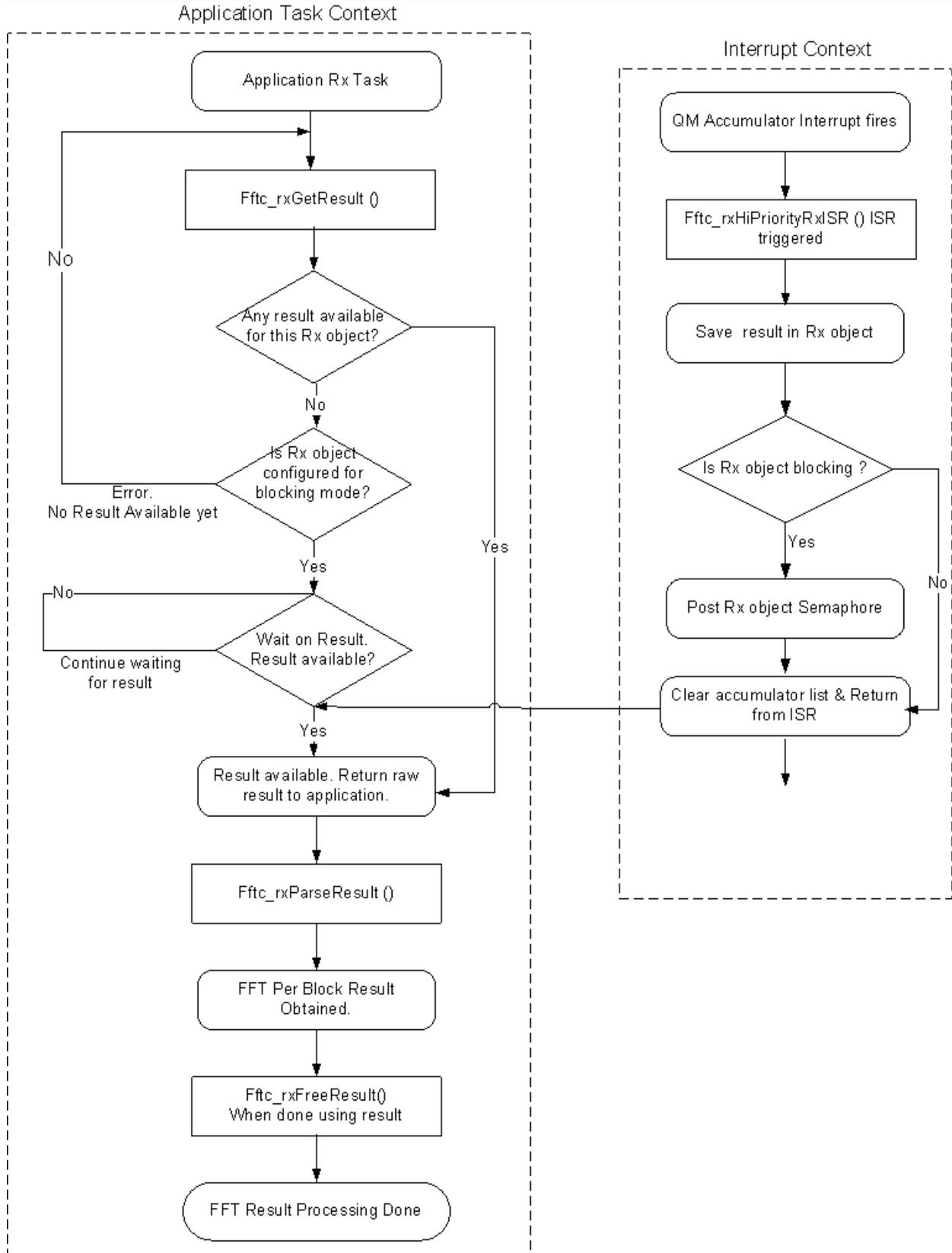


Figure 6: FFTC Receive Path Processing (Interrupt Mode)

5.8.4 De-initialization

Any application that would like to de-initialize and close the driver would have to do so in the following order for proper cleanup:

1. Free any result buffers held with the application using the '*Fftc_rxFreeResult ()*' API.
2. Close any Rx objects opened by the application using '*Fftc_rxClose ()*' API. This API frees up all the resources held by the object, i.e., buffers, descriptors and closes the Rx queue and flow associated with this object.
3. Close any Tx objects the '*Fftc_txClose ()*' API. This API decrements the reference count on the Transmit queue used by the application, and when the reference count reaches zero (when all applications using the Tx queue close the Tx objects using this queue), the driver closes the transmit queue, and cleans up all associated resources.
4. Finally, close the FFTC driver instance opened using the API '*Fftc_close ()*'. This API decrements the reference count on the FFTC peripheral instance state, and frees up the driver object.
5. Finally de-initialize the driver *using* '*Fftc_deinit ()*' API. *When* the reference count on an FFTC instance reaches zero, the driver closes the CPDMA associated with it and resets the FFTC engine's state. Once this is done, no application in the system can use the FFTC CPDMA unless opened again using '*Fftc_init ()*' API.

Note:

Please note that CPPI descriptors setup using '*Fftc_open ()*' API cannot be de-initialized unless a system restart is done. Hence the FFTC de-initialization process is not complete until the entire system is torn down and restarted.

5.9 FFTC OS Abstraction Layer APIs

The FFTC driver defines an OS Abstraction Layer (OSAL) to log, obtain memory, and ensure synchronization to the FFTC driver in an OS independent manner. This section covers all the OSAL APIs that need to be implemented by the System integrator in order to use FFTC driver.

Following are the OSAL APIs that need to be ported by the System integrator for FFTC:

5.9.1 Memory Management

1. *Fftc_osalMalloc:*

- a. This API is called from the FFTC driver to allocate request and result buffers for use by a Rx object / application if it is configured to use “*Host mode*” CPPI descriptors.
- b. It can also be called by the driver to allocate memory for any of the driver’s internal data structures and application handles such as Tx/Rx object handles.
- c. The buffers allocated by this API MUST be global addresses and not core local addresses if the ‘*bGlobalAddress*’ flag is set to 1. Global addresses are required by driver when allocating buffers for host mode descriptors.
- d. By default, this API is implemented as follows in ‘*fftc_osal.h*’:

```
#define Fftc_osalMalloc Osal_fftcMalloc
```

A sample implementation of the API ‘*Osal_fftcMalloc*’ is provided in the ‘*example*’ folder of the FFTC code deliverable.

2. *Fftc_osalFree:*

- a. This API is called from the FFTC driver to free up the memory acquired using the ‘*Fftc_osalMalloc* ()’ API.
- b. The buffer pointer passed to this API is always a global address.
- c. By default, this API is implemented as follows in ‘*fftc_osal.h*’:

```
#define Fftc_osalFree Osal_fftcFree
```

A sample implementation of the API ‘*Osal_fftcFree*’ is provided in the ‘*example*’ folder of the FFTC code deliverable.

5.9.2 Multi-core Synchronization

1. *Fftc_osalMultiCoreCsEnter:*

- a. This API is called from the FFTC driver to obtain a multi-core synchronization lock.

- b. The driver expects that once this lock is obtained that no other process/thread on the current core or on any of the other cores can access the FFTC driver APIs except for the process that has acquired it. The API should be implemented to reflect this.
- c. This API is called from various control path APIs in FFTC driver such as '*Fftc_init ()*', '*Fftc_open ()*' etc to ensure synchronous access to shared data structures.
- d. By default, this API is implemented as follows in '*fftc_osal.h*':

```
#define Fftc_osalMultiCoreCsEnter    Osal_fftcMultiCoreCsEnter
```

A sample implementation of the API '*Osal_fftcMultiCoreCsEnter*' is provided in the '*example*' folder of the FFTC code deliverable.

2. *Fftc_osalMultiCoreCsExit*:

- a. This API is called from the FFTC driver to release a multi-core lock previously obtained using '*Fftc_osalMultiCoreCsEnter ()*' API.
- b. This API is expected to reset the lock so that another process on any of the cores could acquire it to configure and use FFTC library APIs.
- c. By default, this API is implemented as follows in '*fftc_osal.h*':

```
#define Fftc_osalMultiCoreCsExit    Osal_fftcMultiCoreCsExit
```

A sample implementation of the API '*Osal_fftcMultiCoreCsExit*' is provided in the '*example*' folder of the FFTC code deliverable.

5.9.3 Interrupt Lock

1. *Fftc_osalInterruptCsEnter*:

- a. This API is called from the FFTC driver to obtain an interrupt lock, i.e., a lock that can protect the driver against context switches to an interrupt when its manipulating data structures shared between the application process and the interrupt service handler (ISR).
- b. The driver expects that once this lock is obtained, no FFTC interrupt occurs triggering the driver's ISR.

- c. This API is called from the *'Fftc_rxGetResult ()'* to protect the driver maintained list of results posted by ISR against manipulation from ISR at the same time its being accessed by the application in it. This API should disable the interrupts to ensure protection against interrupts.
- d. By default, this API is implemented as follows in *'fftc_osal.h'*:

```
#define Fftc_osalInterruptCsEnter Osal_fftcInterruptCsEnter
```

A sample implementation of the API *'Osal_fftcInterruptCsEnter'* is provided in the *'example'* folder of the FFTC code deliverable.

2. *Fftc_osalInterruptCsExit:*

- a. This API is called from the FFTC driver to release an interrupt lock previously obtained using *'Fftc_osalInterruptCsEnter ()'* API.
- b. This API is expected to enable the interrupts disabled earlier in *'Fftc_osalInterruptCsEnter ()'* API.
- c. By default, this API is implemented as follows in *'fftc_osal.h'*:

```
#define Fftc_osalInterruptCsExit Osal_fftcInterruptCsExit
```

A sample implementation of the API *'Osal_fftcInterruptCsExit'* is provided in the *'example'* folder of the FFTC code deliverable.

5.9.4 Logging

1. *Fftc_osalLog:*

- a. This API is called from the FFTC driver to log useful debug information to the application.
- b. This function can be mapped to an application desired print function to print the info on console or to a *syslog* kind of utility to stream it to a server.
- c. By default, this API is implemented as follows in *'fftc_osal.h'*:

```
#define Fftc_osalLog Osal_fftcLog
```

A sample implementation of the API *'Osal_fftcLog'* is provided in the *'example'* folder of the FFTC code deliverable.

5.9.5 Process notification

1. *Fftc_osalSemCreate:*

- a. This API is called from the FFTC driver to create a software semaphore. This semaphore is used by the driver internally to block on a result for Rx object configured to use interrupts.
- b. This function can be mapped to application's OS semaphore create function.
- c. By default, this API is implemented as follows in *'fftc_osal.h'*:

```
#define Fftc_osalCreateSem Osal_fftcCreateSem
```

A sample implementation of the API *'Osal_fftcCreateSem'* is provided in the *'example'* folder of the FFTC code deliverable.

2. *Fftc_osalSemDelete:*

- a. This API is called from the FFTC driver to delete a semaphore obtained earlier using *'Fftc_osalSemCreate ()'* API.
- b. This function can be mapped to application's OS semaphore delete function.
- c. By default, this API is implemented as follows in *'fftc_osal.h'*:

```
#define Fftc_osalDeleteSem Osal_fftcDeleteSem
```

A sample implementation of the API *'Osal_fftcDeleteSem'* is provided in the *'example'* folder of the FFTC code deliverable.

3. *Fftc_osalPendSem:*

- a. This API is called from the FFTC driver to acquire a software semaphore obtained earlier using *'Fftc_osalSemCreate ()'* API.
- b. This function can be mapped to application's OS semaphore acquire function.
- c. By default, this API is implemented as follows in *'fftc_osal.h'*:

```
#define Fftc_osalPendSem Osal_fftcPendSem
```

A sample implementation of the API '*Osal_fftcPendSem*' is provided in the '*example*' folder of the FFTC code deliverable.

4. *Fftc_osalPostSem*:

- a. This API is called from the FFTC driver to release a software semaphore acquired earlier using '*Fftc_osalPendSem* ()' API.
- b. This function can be mapped to application's OS semaphore release function.
- c. By default, this API is implemented as follows in '*fftc_osal.h*':

```
#define Fftc_osalPostSem Osal_fftcPostSem
```

A sample implementation of the API '*Osal_fftcPostSem*' is provided in the '*example*' folder of the FFTC code deliverable.

5.9.6 Cache Synchronization

1. *Fftc_osalBeginMemAccess*:

- a. This API is called from the FFTC driver before accessing (reading from) any multi-core shared data structures. Since the multi-core shared data structures could be placed in memory accessible across all cores, i.e., either in shared memory or external memory, this API is called from the driver to ensure that the read its performing on the memory block pointer it passed to this API is indeed read from the actual physical memory and not from a local cached copy on the core.
- b. The application is expected to perform the necessary invalidate operation on the memory block passed to ensure that the driver reads correct data from physical memory.
- c. Depending on the level of caches enabled in the system, this API can be mapped to the OS's corresponding Cache invalidate function.
- d. By default, this API is implemented as follows in '*fftc_osal.h*'.

```
#define Fftc_osalBeginMemAccess Osal_fftcBeginMemAccess
```

A sample implementation of the API '*Osal_fftcBeginMemAccess*' is provided in

the *'example'* folder of the FFTC code deliverable.

2. *Fftc_osalEndMemAccess:*

- a. This API is called from the FFTC driver once it's done manipulating a multi-core shared data structure. Since the multi-core shared data structures could be placed in memory accessible across all cores, i.e., either in shared memory or external memory, this API is called from the driver to ensure that the write its performed on the memory block pointer it passed to this API is indeed updated in the actual physical memory too.
- b. The application is expected to perform the necessary write-back operation on the memory block passed to ensure that the data updated by driver is reflected in the actual physical memory too.
- c. Depending on the level of caches enabled in the system, this API can be mapped to the OS's corresponding Cache writeback function.
- d. By default, this API is implemented as follows in *'fft_c_osal.h'*.

<pre>#define Fftc_osalEndMemAccess Osal_fftEndMemAccess</pre>

A sample implementation of the API *'Osal_fftEndMemAccess'* is provided in the *'example'* folder of the FFTC code deliverable.

3. *Fftc_osalBeginDataBufMemAccess:*

- a. This API is called from the FFTC driver before accessing (reading from) a data buffer on the data path. Since the data buffers could be allocated from shared memory or external memory, this API is called from the driver to ensure that the read its performing on the memory block pointer it passed to this API is indeed read from the actual physical memory and not from a local cached copy on the core.
- b. The application is expected to perform the necessary invalidate operation on the memory block passed to ensure that the driver reads correct data from physical memory if the data buffer was allocated from the shared/external memory.
- c. Depending on the level of caches enabled in the system, this API can be mapped to the OS's corresponding Cache invalidate function.
- d. By default, this API is implemented as follows in *'fft_c_osal.h'*.

<pre>#define Fftc_osalBeginDataBufMemAccess Osal_fftBeginDataBufMemAccess</pre>

A sample implementation of the API '*Osal_fftcBeginDataBufMemAccess*' is provided in the '*example*' folder of the FFTC code deliverable.

4. *Fftc_osalEndDataBufMemAccess*:

- a. This API is called from the FFTC driver once it's done manipulating a data buffer on the Tx data path. The only data buffer manipulation done by the driver on Tx data path is to add a control header and any other configuration in the SOP of buffer. Since the data buffer could be allocated from shared/external memory, this API is called from the driver to ensure that the write it's performed on the data buffer memory block pointer it passed to this API is indeed updated in the actual physical memory too.
- b. The application is expected to perform the necessary write-back operation on the memory block passed to ensure that the data updated by driver is reflected in the actual physical memory too. This is required only if the data buffers were indeed allocated from shared/external memory.
- c. Depending on the level of caches enabled in the system, this API can be mapped to the OS's corresponding Cache writeback function.
- d. By default, this API is implemented as follows in '*fftc_osal.h*'.

```
#define Fftc_osalEndDataBufMemAccess Osal_fftcEndDataBufMemAccess
```

A sample implementation of the API '*Osal_fftcEndDataBufMemAccess*' is provided in the '*example*' folder of the FFTC code deliverable.

5.10 FFTC Device Specific Layer APIs

The FFTC driver provides a default device (SoC) specific interrupt configuration in '*device*' folder as a sample. The application can call this API to retrieve the default accumulator Rx queue and channel assignment for the supported SoC.

This is provided as a sample implementation only and must be customized by the system integrator for the SoC on which the application's being used. This API is not used by driver internally.

6 Porting Guidelines

This section covers the porting considerations for FFTC driver:

6.1 Data Types

The FFTC driver by default uses TI's XDC/RTSC data types and is captured in the '*fftc_types.h*' header file. The system integrator must port this header file to use data type definitions that match his system.

6.2 OS AL APIs

The OS AL APIs used by FFTC driver by default are defined in the '*fftc_osal.h*' header file and discussed in Section 5.9 in detail.

The FFTC driver user / system integrator can take either one of the below mentioned approaches in porting the FFTC OS AL to suit his native OS:

1. *Use Pre-built FFTC libraries:*

- a. To use pre-built FFTC libraries, the FFTC driver integrator would have to provide a suitable implementation in his native OS for all the default mapped OS AL APIs, i.e., '*Osal_fftcXxx ()*' defined in '*fftc_osal.h*'.
- b. The application can then link the appropriate version of FFTC pre-built library (choose between Big Endian and Little Endian to match his platform) and can use the FFTC library APIs successfully.
- c. A sample implementation of this approach has been provided in the '*example*' directory of the FFTC deliverable.
- d. This approach is better suited if the customer would like to use the TI provided pre-built FFTC libraries right out of the box.
- e. The demerit of this approach would be that, there is one extra level of indirection in getting to the actual OS AL API implementation from the FFTC driver APIs.

For example:

FFTC driver APIs use *Fftc_osalMalloc* internally, which is in turn mapped at pre-compilation to \rightarrow *Osal_fftcMalloc* that contains actual implementation of the API in native OS.

2. *Rebuild FFTC from sources:*

- a. To do so, the system integrator must make a copy of the *fftc_osal.h* header file and modify it to directly map the *Fftc_osalXxx* APIs used by the FFTC driver APIs to his inline native OS calls.

For Example:

The example shows a sample mapping assuming the OS used is BIOS. Here the Memory allocation API used by FFTC driver, i.e., *Fftc_osalMalloc* is mapped to the corresponding memory allocation API from BIOS 6 *Memory_alloc ()*.

```
#defineFftc_osalMalloc      Memory_alloc
```

- b. Once all the FFTC OSAL APIs discussed in Section 5.9 are implemented as shown above using native OS calls, the integrator can include his new header file with his OS definitions and recompile the FFTC sources.
- c. The *test* directory provides a sample implementation of this approach using BIOS.
- d. This approach eliminates the extra OSAL API indirection that was there in the previous approach and hence is better optimized for performance.
- e. However, the downside of this approach is that the customer would have to rebuild the FFTC library from sources.

6.3 Configuration Options

The FFTC driver exposes a compile time configuration option “FFTC_DRV_DEBUG” to control parameter validations on all fast path APIs.

The driver libraries by default are built with this option turned OFF, i.e., no parameter validations built into the fast path HLD and LLD APIs. If the library user would like to build a debug version of the library with parameter validations, he could include the driver sources into his application project, include this option in compiler’s build definitions and rebuild his project. Alternatively, the customer can modify the library build files to include this option and re-build the library from command line to enable parameter validations in driver for the application.

7 Future Extensions